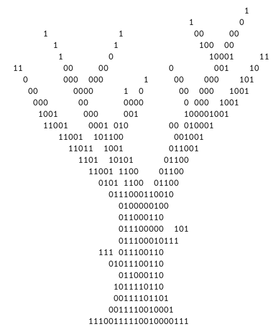
# CSE 143X

## Accelerated Computer Programming I/II



# Binary Trees



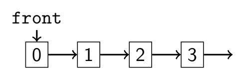
## Outline



## Back To LinkedLists



Consider the following standard LinkedList:




**Recall the definition of a ListNode**

```
public class Node {
   public int data;
   public Node next;

   public Node(int data, Node next) {
      this.data = data;
      this.next = next;
   }
}
```

**What if we added more fields?**

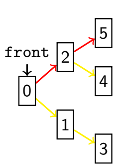
- Multiple data fields?
- Multiple "next" fields?

## Back To LinkedLists



**Nodes with Multiple next Fields**

```
public class Node {
   public int data;
   public Node next1;
   public Node next2;

   public Node(int data, Node next1, Node next2) {
      this.data = data;
      this.next1 = next1;
      this.next2 = next2;
   }
}
```



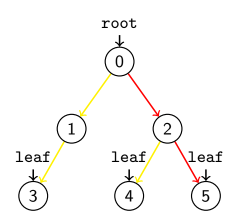

(yellow is next2; red is next1)

## Introducing Trees



**Binary Trees**

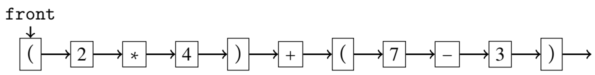
```
public class Node {
   public int data;
   public Node left;
   public Node right;

   public Node(int data, Node left, Node right) {
      this.data = data;
      this.left = left;
      this.right = right;
   }
}
```




(red is right; yellow is left)

## Back To `LinkedLists` 

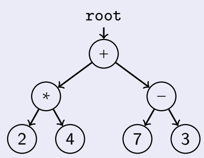
Consider the following `LinkedList` of a mathematical expression:




**What's bad about it?**

- It doesn't really help us with the structure
- Looking at it doesn't really show us what's going on

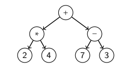
**What about this structure instead?**



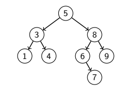

**Now we can see the order of operations much more clearly!**

## Uses of Trees 

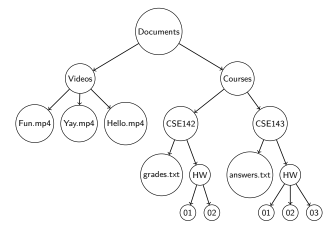
- Parsing (Programming Languages, Math, etc.)



- Implementing `TreeSet`

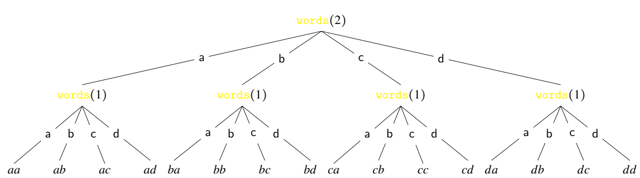


- Directory File Structure




## More Uses of Trees 

- Recursive Trees (including things like games of Tic-Tac-Toe)




- Compression (this will be your last assignment!)

## Printing A `LinkedList` (Again) 

```java
public void print() {
   Node current = this.front;
   while (current != null) {
      System.out.print(current.data + " ");
      current = current.next;
   }
}
```

We'd like to figure out how to print trees. Since `LinkedLists` are "simpler versions of trees", they might help.

**How do we go in every direction in a tree?**

**USE RECURSION!**

## Printing a `LinkedList` Recursively 

**To print a `LinkedList`...**

- Print the **front** of the list
- Print the **next** of the list (recursively)

**Code**

```java
public void print() {
   print(this.front);
}

public void print(Node c) {
   if (c != null) {
      System.out.print(c.data + " ");
      print(c.next);
   }
}
```

## Printing a `Tree` Recursively 

**To print a `BinaryTree`...**

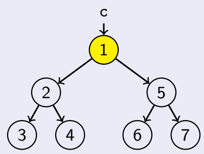
- Print the **root** of the tree
- Print the **left** of the tree (recursively)
- Print the **right** of the tree (recursively)

**Code**

```java
public void print() {
   print(this.root);
}

public void print(Node c) {
   if (c != null) {
      System.out.print(c.data + " ");
      print(c.left);
      print(c.right);
   }
}
```

```java
public void print(Node c) { // c = ①
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace




OUTPUT

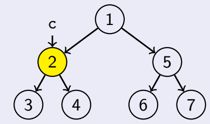
```
>> 1
```



```java
public void print(Node c) { // c = ①
    if (c != null) {
```

```java
public void print(Node c) { // c = ②
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace




OUTPUT

```
>> 1 2
```

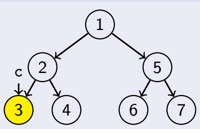


```java
public void print(Node c) { // c = ①
    if (c != null) {
```

```java
public void print(Node c) { // c = ②
    if (c != null) {
```

```java
public void print(Node c) { // c = ③
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace




OUTPUT

```
>> 1 2 3
```



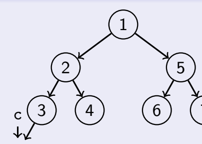
```java
public void print(Node c) { // c = ①
    if (c != null) {
```

```java
public void print(Node c) { // c = ②
    if (c != null) {
```

```java
public void print(Node c) { // c = ③
    if (c != null) {
```

```java
public void print(Node c) { // c = null
    if (c != null) {           // c is null!
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace




OUTPUT

```
>> 1 2 3
```

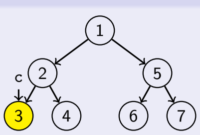


```java
public void print(Node c) { // c = ①
    if (c != null) {
```

```java
public void print(Node c) { // c = ②
    if (c != null) {
```

```java
public void print(Node c) { // c = ③
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace




OUTPUT

```
>> 1 2 3
```



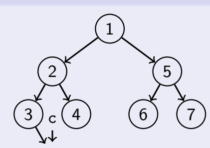
```java
public void print(Node c) { // c = ①
    if (c != null) {
```

```java
public void print(Node c) { // c = ②
    if (c != null) {
```

```java
public void print(Node c) { // c = ③
    if (c != null) {
```

```java
public void print(Node c) { // c = null
    if (c != null) {           // c is null!
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace



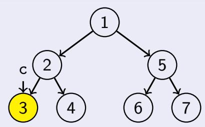

OUTPUT

```
>> 1 2 3
```

## Printing a Tree Example


```
public void print(Node c) { // c = ①
    if (c != null) {
```
```
public void print(Node c) { // c = ②
    if (c != null) {
```
```
public void print(Node c) { // c = ③
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace

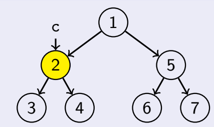



OUTPUT
```
>> 1 2 3
```

## Printing a Tree Example


```
public void print(Node c) { // c = ①
    if (c != null) {
```
```
public void print(Node c) { // c = ②
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace

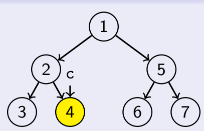



OUTPUT
```
>> 1 2 3
```

## Printing a Tree Example


```
public void print(Node c) { // c = ①
    if (c != null) {
```
```
public void print(Node c) { // c = ②
    if (c != null) {
```
```
public void print(Node c) { // c = ④
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace

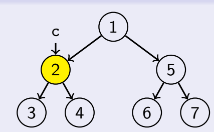



OUTPUT
```
>> 1 2 3 4
```

## Printing a Tree Example


```
public void print(Node c) { // c = ①
    if (c != null) {
```
```
public void print(Node c) { // c = ②
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace

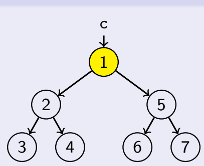



OUTPUT
```
>> 1 2 3 4
```

## Printing a Tree Example


```
public void print(Node c) { // c = ①
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace

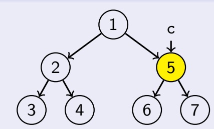



OUTPUT
```
>> 1 2 3 4
```

## Printing a Tree Example


```
public void print(Node c) { // c = ①
    if (c != null) {
```
```
public void print(Node c) { // c = ⑤
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace

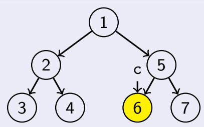



OUTPUT
```
>> 1 2 3 4 5
```

## Printing a Tree Example


```
public void print(Node c) { // c = ①
    if (c != null) {
public void print(Node c) { // c = ⑤
    if (c != null) {
public void print(Node c) { // c = ⑥
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace



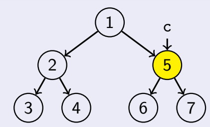

OUTPUT

```
>> 1 2 3 4 5 6
```

## Printing a Tree Example


```
public void print(Node c) { // c = ①
    if (c != null) {
public void print(Node c) { // c = ⑤
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace



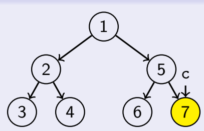

OUTPUT

```
>> 1 2 3 4 5 6
```

## Printing a Tree Example


```
public void print(Node c) { // c = ①
    if (c != null) {
public void print(Node c) { // c = ⑤
    if (c != null) {
public void print(Node c) { // c = ⑦
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace



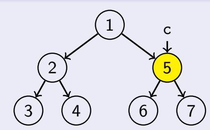

OUTPUT

```
>> 1 2 3 4 5 6 7
```

## Printing a Tree Example


```
public void print(Node c) { // c = ①
    if (c != null) {
public void print(Node c) { // c = ⑤
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace



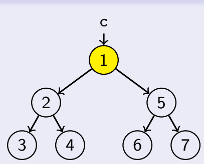

OUTPUT

```
>> 1 2 3 4 5 6 7
```

## Printing a Tree Example


```
public void print(Node c) { // c = ①
    if (c != null) {
        System.out.print(c.data + " ");
        print(c.left);
        print(c.right);
    }
}
```

Trace




OUTPUT

```
>> 1 2 3 4 5 6 7
```

## Tree Traversals


Pre-Order Traversal

```
public void print(Node c) {
    if (c != null) {
        System.out.print(c.data + " "); // print
        print(c.left);                  // left
        print(c.right);                 // right
    }
}
```

In-Order Traversal

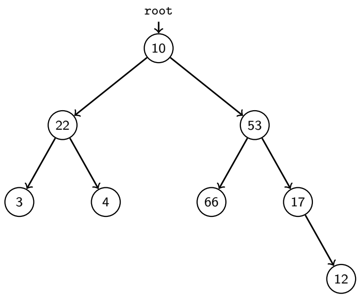
```
public void print(Node c) {
    if (c != null) {
        print(c.left);                  // left
        System.out.print(c.data + " "); // print
        print(c.right);                 // right
    }
}
```

Post-Order Traversal

```
public void print(Node c) {
    if (c != null) {
        print(c.left);                  // left
        print(c.right);                 // right
        System.out.print(c.data + " "); // print
    }
}
```

## Tree Traversal Example 

Consider the following binary tree:



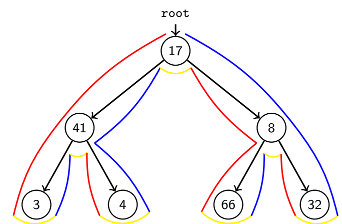

Compute the Pre-Order, In-Order, and Post-Order Traversals:

- Pre-Order: **10, 22, 3, 4, 53, 66, 17, 12**
- In-Order: **3, 22, 4, 10, 66, 53, 17, 12**
- Post-Order: **3, 4, 22, 66, 12, 17, 53, 10**

## Traversal Trick 

**To Quickly Generate A Traversal**

- Trace a path around the tree
- As you pass a node on the proper side, process it:
  - Pre-Order: left
  - In-Order: bottom
  - Post-Order: right




## Binary Tree `method` 

Binary Tree methods are just normal recursive functions. The base case/recursive calls will always be similar.

**Writing a Binary Tree Method**

- The base case is `current == null`.
- First recursive case is `method(current.left)`
- Second recursive case is `method(current.right)`

```
public type method(...) {
   return method(this.root, ...);
}
private type method(TreeNode current, ...) {
   if (current == null) { /* DO BASE CASE */ }

   // Do the left recursive case:
   type leftResult = method(current.left, ...);

   // Do the right recursive case:
   type rightResult = method(current.right, ...);

   /* Use the left and right results... */
   return ...;
}
```

## Binary Tree `contains()` 

**`contains()`**

Write a method, in the `IntTree` class, called `contains()`:

`public boolean contains(int value);`

that returns true if the tree contains `value` and false otherwise.

```
public boolean contains(int value) {
   return contains(this.root, value);
}
private boolean contains(IntTreeNode current, int value) {
   /* If the tree is null, it definitely doesn't contain value... */
   if (current == null) { return false; }

   /* If current *is* value, we found it! */
   else if (current.data == value) { return true; }

   else {
      boolean leftContainsValue = contains(current.left, value);
      boolean rightContainsValue = contains(current.right, value);
      return leftContainsValue || rightContainsValue;
   }
}
```

## Some Tree Tips! 

- Trees are just generalized `LinkedLists`. So, all of the things you learned about references with `LinkedLists` are going to apply to trees as well
- Almost all the tree methods you write will be recursive (and will have a private helper that takes in the root)
- Make sure you understand all the traversals; the trick can be very useful.